managerial posts and conference chairmanships in both the American Association for Artificial Intelligence (AAAI) and the International Joint Conference on Artificial Intelligence (IJCAI).

Several KSL faculty and former students have received significant honors. In 1976, Ted Shortliffe received the Association of Computing Machinery Grace Murray Hopper award. In 1977, Doug Lenat was given the IJCAI Computers and Thought award, and in 1978, Ed Feigenbaum received the National Computer Conference Most Outstanding Technical Contribution award. In 1979 and 1981, Ted Shortliffe's book *Computer-Based Medical Consultation: MYCIN* was identified as the most frequently cited work in the IJCAI proceedings. In 1982, Doug Lenat won the Tioga prize for the best AAAI conference paper while Mike Genesereth received honorable mention. In 1983, Ted Shortliffe was named a Kaiser Foundation faculty scholar, and Tom Mitchell received the IJCAI Computers and Thought award. In 1984, Ed Feigenbaum was elected a fellow of the American Association for the Advancement of Science (AAAS), and he and Ted Shortliffe were elected fellows of the American College of Medical Informatics(ACMI). Larry Fagan was elected a fellow of ACMI in 1985. In 1986, Ed Feigenbaum was elected to the National Academy of Engineering and in 1987, Ted Shortliffe was elected to the Institute of Medicine of the National Academy of Sciences. The American Association for Medical Systems and Informatics Young Investigator Award for Research in Medical Knowledge Systems was presented to Glenn Rennels in 1988 and to Mark Musen in 1989.

KSL Research Environment

Funding—The KSL is supported solely by sponsored research and gift funds. We have had funding from many sources, including DARPA, NIH/NLM, ONR, NSF, NASA, and private foundations and industry. Of these, DARPA and NIH have been the most substantial and long-standing sources of support. All, however, have made complementary contributions to establishing an effective overall research environment that fosters interchanges at the intellectual and software levels and that provides the necessary physical computing resources for our work.

Computing Resources—Under the Symbolic Systems Resources Group, the KSL develops and operates its own computing resources tailored to the needs of its individual research projects. Current computing resources are a networked mixture of personal workstations, Lisp workstations, and central host computers and network utility servers, reflecting the evolving hardware technology available for AI research. Our central host is currently a Sun 4/280 running Sun Unix 4.0 (this is the core of the national SUMEX biomedical computing resource). It provides a central service for remote network access, electronic mail storage and routing, large-scale file storage, and printer spooling services. Increasingly, computing functions, such as electronic mail reading and composition, text processing, and information retrieval, are being moved to distributed user workstations. Our Lisp workstations include 34 Texas Instruments Explorers, 2 Symbolics 3600-

series machines, 3 SUN 3/75 workstations, and 4 NeXT machines. Much of the routine computing is done with 80 Apple Macintosh II computers, 15 of which have Texas Instruments microExplorer Lisp co-processor boards. Network printing, file storage, Internet gateway, and terminal interface services are provided by dedicated machines including a VAX 11/750, a SUN 3/180, and numerous special-purpose microprocessor systems. These facilities are integrated with other computer science resources at Stanford through an extensive Ethernet and to external resources through the ARPANET, TELENET, and the BARRNet (Bay Area Regional Research Network) link to the NSFNet. Funding for these resources comes principally from DARPA and NIH and hardware vendor gifts.

Appendix B: Lisp Performance Studies

Performance of Two Common Lisp Programs On Several Systems (Report KSL 89-02)

by Richard Acuff

Abstract

To assist in the evaluation of Lisp platforms for the Stanford University Knowledge Systems Laboratory, 22 Common Lisp implementations were benchmarked. Run time and compilation time data on two moderate-sized application programs are presented, along with data on the effect of compiler optimization levels and on the impact of display I/O on run time. For these Lisp benchmarks, several systems did not rank where we expected them based on speed ratings using other conventional measures. Also, the rankings of machines by Lisp speed differed for the two programs we tested The data indicate that the performance of Lisp systems is very application dependent. Software environment should play at least as strong a role in machine selection as performance benchmarks.

1. Introduction

At Stanford University's Knowledge Systems Laboratory (KSL), a large amount of software is written in Lisp. Thus, the performance of Lisp systems is often crucial to the productivity of the lab. In order to assist us in understanding the performance of different Lisp systems, we have undertaken an informal survey of 22 Common Lisp implementations using two software packages developed in the KSL. The main goal of this survey was to understand the execution speed performance of systems that we might use in the KSL for research and development or dissemination of research results. Secondary goals were to evaluate the effect of compiler optimizer settings on execution speed and to evaluate the effect of reducing the amount of output on execution speed.

There have been a number of projects to measure the performance of Lisp systems. Gabriel's work [Gabriel 1985] is probably the best known, and is the origin of the so-called "Gabriel Benchmarks", a set of small test programs for measuring specific aspects of Lisp system performance. The Gabriel benchmarks are extremely valuable, for people trying to compare Lisp systems, if used knowledgeably. However, the aspects of a Lisp system stressed by a particular program are often difficult to determine so that it is usually best, where possible, to run that program on the systems in question rather than attempting to dissect the program and forecast its performance analytically. Also, with the advent of numerous implementations of Common Lisp [Steele 1984], we can now use much larger test programs without the bother and uncertainty of porting between dialects.

In this survey we have focused on execution speed which has long been an important criterion for comparing computer systems. The first comparison of

two systems solving the same problem (benchmarking) was probably made shortly after the creation of the second computer, and benchmarking has been a primary differentiator among computer systems ever since. However, execution speed benchmarks are only one aspect of the performance of systems, especially Lisp systems. Issues like programming and user environments, compatibility with other systems, the ability to handle "large" problems, and cost (hardware, software, and human) must also be considered, and, given a machine that is "fast enough", these other issues will almost always be the overriding factor.

Descriptions of the programs used in this evaluation are given in Section 2. A description of the methodology used in performing the tests is in Section 3, and information about the Lisp systems tested is in Section 4. Data on the execution speed of the test programs are presented in Section 5, followed by compilation speed data and a comparison between compilation speed and execution speed in Section 6. The effect of choosing various values for the SPEED and SAFETY options of the OPTIMIZE declaration on the BB1 system are discussed in Section 7. The effect of reducing the screen output of the SOAR benchmark is presented in Section 8. Details of the test procedures and descriptions of the systems tested are in the appendices.

2. Test Software

The software systems used in these tests were SOAR [Laird 1987] and the BB1 blackboard core [Hayes-Roth 1985 and Hayes-Roth 1988]. These test programs were chosen primarily because they are implemented in pure Common Lisp, making them extremely portable[1]. Both are systems in daily use in the KSL but represent two distinct research directions in terms of program function and structure. These systems were initially developed in environments other than those tested, and no attempt was made to optimize their performance for any of these tests. Neither of these systems is an intensive user of numeric computation.

A copy of the Common Lisp source code used for these tests may be obtained from the author by sending U.S. Mail to "Richard Acuff, Stanford KSL, 701 Welch Road, Bldg. C, Stanford, CA 94305" or electronic mail to "acuff@SUMEX-AIM.Stanford.EDU".

---

1 There were one or two small porting difficulties that were traced to problems in the test code which had to be fixed. For instance, many systems allow `(INTERN "NAME" 'USER)` where others require `(INTERN "NAME" (FIND-PACKAGE "USER"))`. Also we were unable to get SOAR to work in either versions 1.0 or 1.1 of Allegro Common Lisp for the Mac II due to unexplained software hangs so it is omitted from SOAR-related charts.

2.1. SOAR

SOAR is a heuristic-search based general problem solving architecture developed by Paul Rosenbloom, *et. al.* See [Laird 1987] for more information on the SOAR system.

All test runs of SOAR were done solving an eight-puzzle problem in one of three modes: Mode A (simply solve the problem), Mode B (solve the problem while "chunking" or "learning"), and Mode C (solve the problem after having "learned" in Mode B).

An "eight puzzle" is a common children's game with 8 tiles, numbered 1 to 8, on a 3 by 3 grid such that a tile adjacent to the empty place can be pushed into it. "Solving the eight-puzzle problem" consists of producing a series of tile moves such that, from a given arbitrary starting configuration, the eight puzzle ends up with all the tiles in numerical order, reading from the upper left around the puzzle clockwise, with the empty place in the middle.

The version of SOAR used was 4.4.4, dated April 19, 1987. It consists of 1 large LISP source file and 2 small SOAR files containing productions for solving the eight-puzzle problem. The LISP source is 10,661 lines (280,050 characters) of lightly commented code.

2.2. BB1

BB1 is a blackboard-based problem solving architecture developed by Barbara Hayes-Roth. For more information on the BB1 blackboard core, see [Hayes-Roth 1985]. For further information on BB1, see [Hayes-Roth 1988]. All references to BB1 in this document refer only to the "core" blackboard parts of the system and do not include any other layers of the problem solving architecture or the user interface, as these components are not in pure Common Lisp. All test runs of BB1 went through three cycles of adding 10 items to the blackboard, accessing those 10 items, and then deleting them.

The version of BB1 used was 1.2. The LISP source used consists of 10 files ranging from 36 lines (814 characters) to 3,396 lines (107,528 characters) of lightly commented code, with a total of 8,722 lines (295,199 characters) of code.

3. Methodology

All the tests were performed in as near to a "normal" working environment as could be achieved. We tried to duplicate the working conditions that a researcher would likely have both in hardware and software. Where possible we selected test machines configured with the amount of memory, amount and type of disk, type of display, etc. that a typical developer would purchase and use. We ran the software in a way that a developer using the system would probably use it. Thus, if it was normal to run with garbage collection enabled, under a window system, within an editor, or in a multi-programming environment, then that was done. For instance, Sun machines were tested under *SunView* with a couple of *perfmeter*s running. The HP

machine was tested while running in *GnuEmacs* on *X.10*. MIT-style Lisp machines were run with all networking and other background processing on, and no special process priority. No expert tuning or system configuration was done beyond what the tester could do by reading over the user documentation. All systems were tested in single-user mode, which is the way those tested are normally used for Lisp work.

We feel that although this methodology results in less repeatable and less explainable results, it gives a good approximation to what the end user will experience. Where time allowed, multiple runs were made to ensure accurate readings. Unfortunately the collection of the raw data (i.e. arranging for machine access and making the timed runs) proved to be an extremely time consuming process, taking a day or more for some of the systems, so the information in this report was collected over a long period of time (October, 1987 to January 1989) and some of the data may be dated by now.

The procedures used for running the tests are fully described in Appendix B. The TIME macro was used to collect timing information. Most times were recorded to the nearest second. When reported by the TIME macro, some extra information, usually relating to paging, memory management, "kernel" time, etc., were recorded, but are not analyzed here. If several runs were made, only the best number is reported herein for the sake of brevity. Wherever possible, source files were stored on local disks (for the Sun 3/75 systems the files were on a Sun 3/180 NFS server on the same subnet).

4. Systems Under Test

The systems that we tested were chosen based on their availability to the testers as well as their suspected usefulness in future KSL programming efforts. All of the systems tested were workstations, as we were not able to obtain access to mainframe systems. It is also the case that workstations, with their bit-mapped displays and dedicated processors, currently provide the best Lisp development environments, in our opinion, and thus were more interesting to us.

A mnemonic code is used for each of the 22 systems. Usually the code is the model of the machine except where there is more than one Lisp for a machine (as in the case of the Sun 3/75) in which case a letter is prefixed to indicate the Lisp being used. Table 1 gives a mapping between codes and machine types. See Appendix A for detailed descriptions of system configurations.

5. Execution Speed

Most of the tables and charts in this report refer to elapsed-times (wall-clock time) in seconds. Most of the tables and charts have the system types ordered according to what seems to be the most interesting comparison. We have attempted to group systems of allegedly comparable performance (according to our perception formed from talking to vendor representatives, talking to other users, reading reports, etc.)

Code	Test Date	System Type
3/260	Summer 1988	Sun 3/260 with Lucid Lisp[1]
3/60	Summer 1988	Sun 3/60 with Lucid Lisp
386	Spring 1988	Compaq 386 with Lucid Lisp
386T	Spring 1988	Compaq 386 portable with Lucid Lisp
4/260	Summer 1988	Sun 4/260 with Lucid Lisp
4/280	Winter 1988	Sun 4/280 with Lucid Lisp
DEC-II	Fall 1987	DEC MicroVax II with VaxLisp
DEC-III	Fall 1987	DEC MicroVax III with VaxLisp
E-3/75	Fall 1987	Sun 3/75 with Franz Extended Common Lisp
EXP1	November 1988	Texas Instruments Explorer I
EXP2	November 1988	Texas Instruments Explorer II
EXP2+	November 1988	Texas Instruments Explorer II Plus
F-4/280	January 1989	Sun 4/280 with Franz Allegro Common Lisp
HP	Fall 1987	Hewlett Packard 9000/350
K-3/75	Fall 1987	Sun 3/75 with Kyoto Common Lisp
L-3/75	Summer 1988	Sun 3/75 with Lucid Lisp
Mac2	Spring 1988	Apple Macintosh II with Allegro Common Lisp
Maci	December 1988	Symbolics MacIvory
mX	November 1988	Texas Instruments microExplorer
RT	Spring 1988	IBM RT/APC with Lucid Lisp
Sym	Winter 1988	Symbolics 3645
XCL	Winter 1988	Xerox 1186

Table 1: Mapping between codes and system types

It is worth noting that on almost all of the systems tested, virtual memory paging was a negligible part of the overall run time for the tests. Nor was it a very significant factor during compilation. In general, we do not expect this to be true for most production systems. Indeed, we would not be surprised if paging time were a major component of overall run time for most systems.

5.1. BB1

The data for the run times of the BB1[2] tests are given in Table 2. Figure 1 shows the data graphically.

---

1 The Lucid and Franz Extended Common Lisp products tested are versions prior to multi-programming within the Lisp and prior to the inclusion of generation-based scavenging garbage collection in those systems. The Allegro Common Lisp was not tested with multiprogramming enabled.

2 These times are for default settings of the SPEED and SAFETY optimization qualities discussed in Section 7.

Code	Run Time	Code	Run Time
Exp2	27	RT	75
Exp2+	17	DEC-III	63
4/260	56	Exp1	87
4/280	34	3/60	73
-4/280	56	L-3/75	90
386	47	E-3/75	211
386T	54	K-3/75	96
mX	33	HP	115
Maci	129	DEC-II	207
Sym	111	XCL	559
3/260	62	Mac2	254

Table 2: Run times for BB1

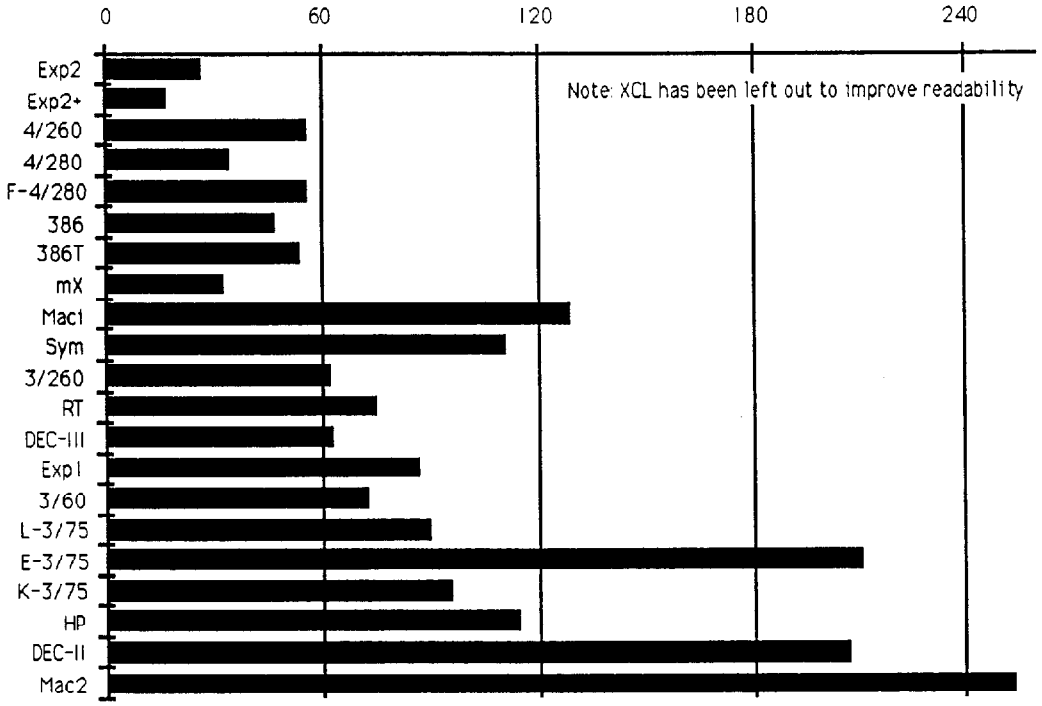

Figure 1: BB1 Run (sec)

Systems that are marketed as comparable generally came out close to each other with the following notable exceptions:

- There was a significant difference between the 4/280 and the 4/260. Even though the 4/260 had more memory, similar disk, more tuning effort, and was tried with several later versions of Lisp it was consistently slower than the 4/280 tested earlier. We are at a loss to explain this discrepancy.

It is also worth noting that, except for VaxLisp, Lucid Lisp seemed the most difficult to tailor to a particular machine when it was being installed.

- The DEC machines seem to be poor at running Lisp even though they are usually thought of as competitive when running FORTRAN or C.
- The microExplorer (mX) did better than expected probably because its weak point, paging, was not stressed by this test.
- The much older Franz Lisp (E-3/75) did relatively poorly compared to Lucid Lisp on the 3/75, but the newer version on the Sun 4 did well relative to the somewhat older Lucid lisp on the Sun 4.
- XCL was over twice as slow as the nearest competitor.
- For unknown reasons the Symbolics machines were slower than expected. The MacIvory was a bit over 4 times slower than the microExplorer and the 3645 was slower than the Explorer I.

5.2. SOAR

The data for the SOAR run tests are given in Table 3 and presented graphically in Figure 2. The figures are for the sum of the A, B, and C modes[1].

Once again most systems fit where expected with the following notes:

- The Lucid Sun 4's are somewhat faster than the TI Explorer II for the SOAR test whereas the opposite was true for the BB1 test.
- XCL and DEC-II were over twice as slow as the nearest other system.

Code	Run Time	Code	Run Time
Exp2	94	RT	177
Exp2+	62	DEC-III	454
4/260	58	Exp1	369
4/280	82	3/60	187
-4/280	120	L-3/75	278
386	126	E-3/75	484
386T	151	K-3/75	697
mX	154	HP	219
Maci	339	DEC-II	1851
Sym	193	XCL	1519
3/260	154	Mac2	No data (see footnote 1)

Table 3: Aggregate Run Times for SOAR

[1] The A and C mode figures are for the "no trace" configuration as described in Section 8.

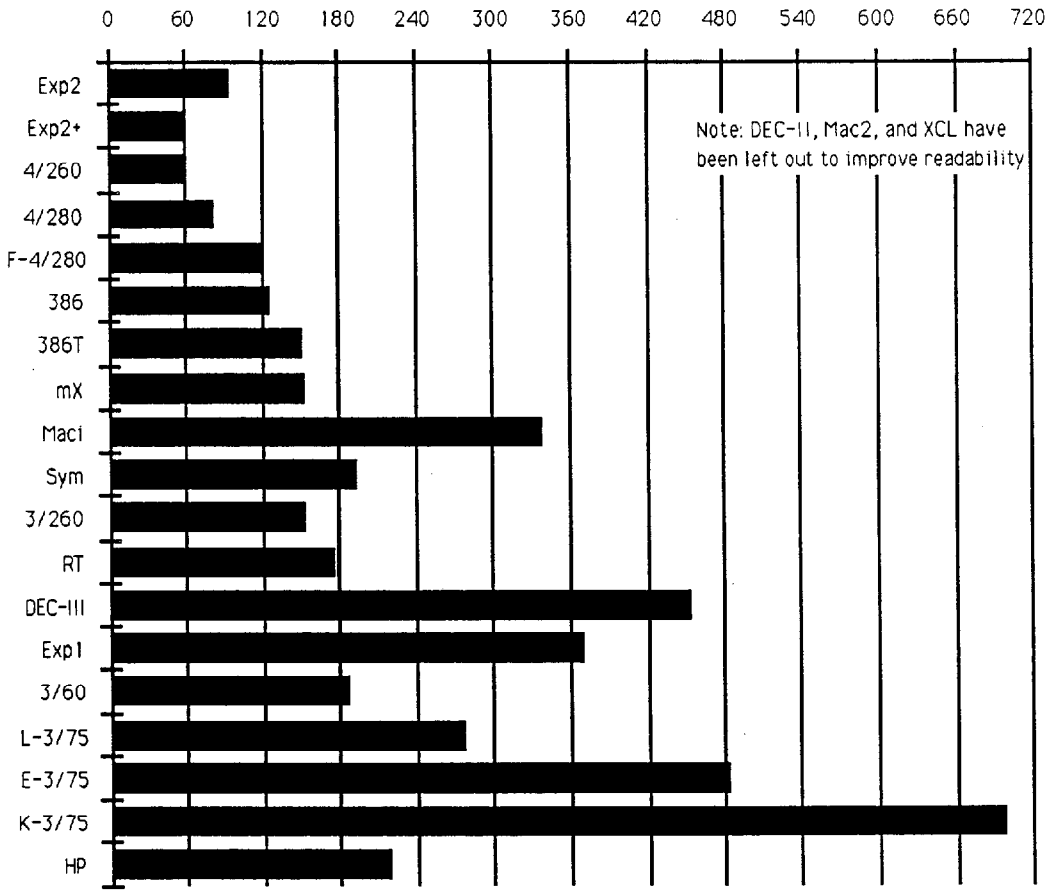

Figure 2: Sum of SOAR run times (sec)

5.3. Normalized Run Times

A given machine, call it *A*, may have run the SOAR test faster than another machine, *B*, while *B* was faster for BB1. Figure 3 depicts this difference. For both BB1 and SOAR the run times have been normalized by dividing the run time by the average of the run times for all the machines, leaving out DEC-II, Mac2, and XCL to improve readability.

Lucid Lisp seemed to perform relatively better with SOAR than with BB1 in all cases, while VaxLisp and, to a much lesser extent, the dedicated Lisp machines, seemed to do better with BB1.

There are many possible explanations for these variations, but trying to analyze each of them was well beyond the scope of this study. The reasons are most likely a result of differences among implementations in the efficiency of various operations, some of which are used by SOAR but not by BB1 and *vice versa*. For instance, SOAR might make heavy use of hashing

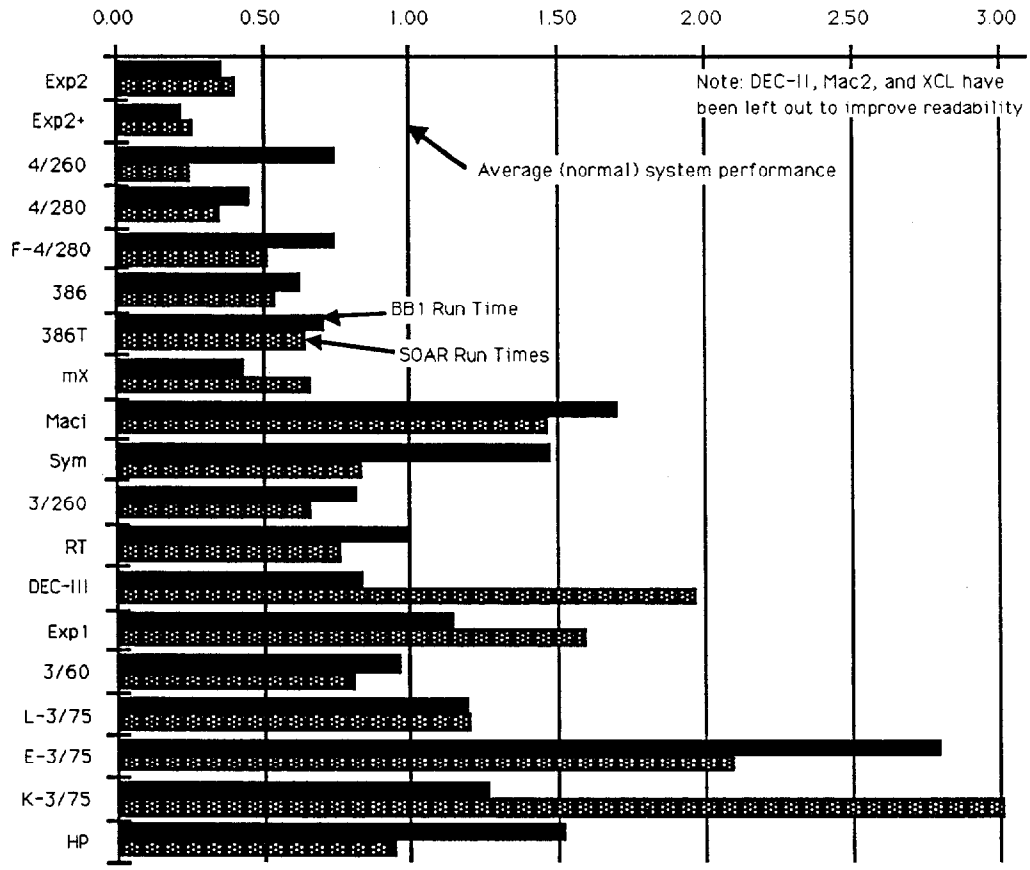

Figure 3: Normalized Run Times (time/average_time)

while BB1 makes heavy use of list primitives, or one system might include a large number of SETQ operations while the other might be more applicative in nature. The developers of SOAR and BB1 do not currently have information on the aspects of the Lisp systems stressed by their software.

6. Compilation Speed

Developers and researchers must worry about how fast their programs compile as well as how fast they run. SOAR and BB1 compilation times are given in Table 4 and Figure 4.

Figures 5 compares run time with compile time. The ratio of compilation time to run time is shown. A system with a high rating spends relatively more time compiling than running. The absolute value of these numbers have little meaning. They are only useful for comparing systems.

Code	SOAR	BB1	Code	SOAR	BB1
Exp2	132	89	RT	574	586
Exp2+	78	76	DEC-III	423	633
4/260	307	324	Exp1	520	327
4/280	523	482	3/60	569	551
F-4/280	535	264	L-3/75	1040	919
386	386	355	E-3/75	450	444
386T	479	416	K-3/75	1365	1234
mX	152	186	HP	237	235
Maci	906	950	DEC-II	1227	1774
Sym	252	257	XCL	1800	1927
3/260	687	540	Mac2	0	349

Table 4: Compilation Times

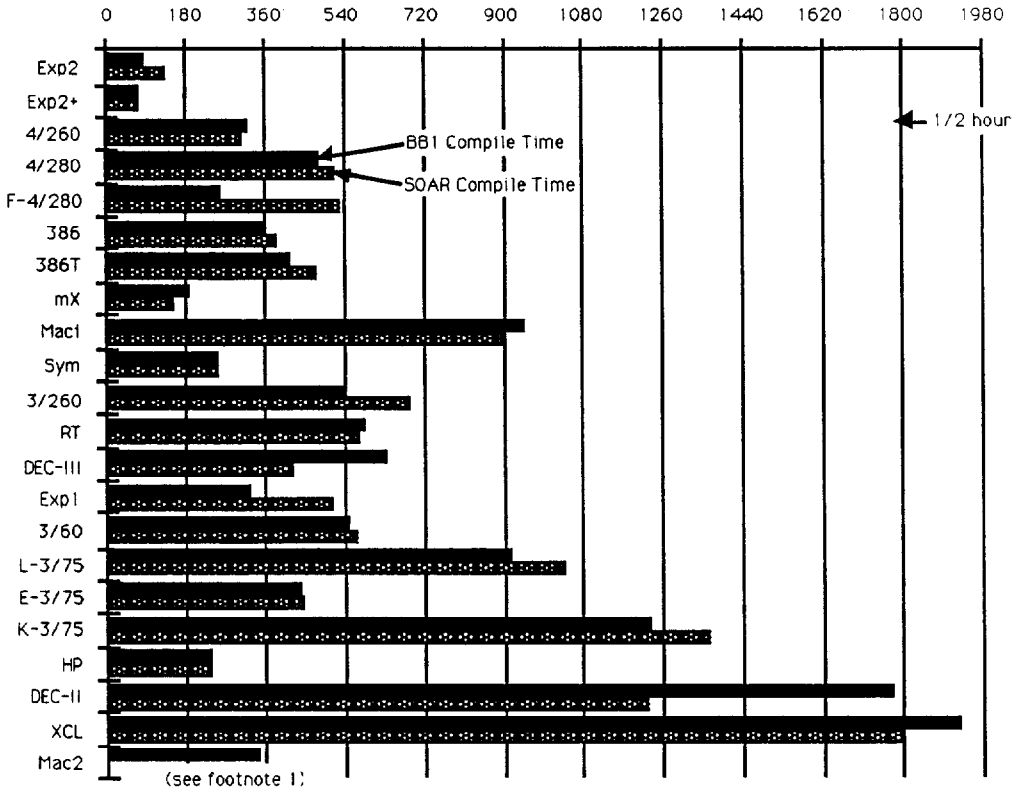

Figure 4: Compilation Time (sec)

As one might expect, the specially microprogrammed Lisp machines had relatively fast compilers. Some machines with run times slower than predicted spent relatively less time compiling. For example, the VaxLisp compiler was relatively fast, but generated very slow code. The Lucid compiler seemed to take a long time but generated fast code. The Allegro

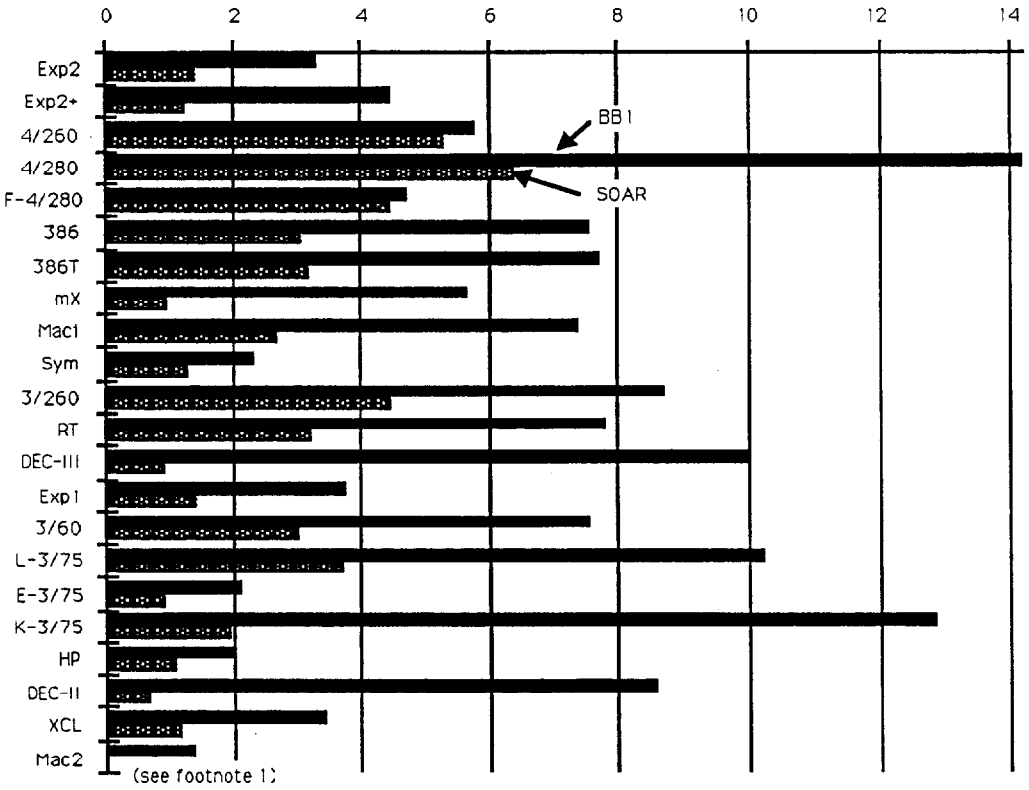

Figure 5: Relative Performance of Compiler
(Compile_Time/Run_Time)

Common Lisp for the Mac II took little time but still somehow generated impressively fast code for BB1.

7. Effect of OPTIMIZE Settings on BB1

The OPTIMIZE declaration is a way of controlling the behavior of a Common Lisp compiler. Two of the most significant qualities thus controlled are SPEED and SAFETY. Each of these can be set to an integer from 0 to 3. A high setting for SPEED tells the compiler that fast running code is desired, which typically enables various optimizations. The Common Lisp specification doesn't require any optimizations or even that they necessarily be controlled by this setting, but many current implementations switch on optimizers such as dead code eliminators, tail and mutual recursion eliminators, fancy register allocators, and facilities to take advantage of type declarations. The SAFETY quality is somewhat less well understood. It has little to do with the "safety" of the program since a correct Common Lisp program is still required to run correctly if SAFETY is low, but it has an impact on the debuggability of the program. A high SPEED and low SAFETY may allow, for instance, disabling number-of-arguments checking to allow

faster function calls on some architectures, or type checking on system functions (such as CAR or SETQ) might be disabled. Kyoto Common Lisp (KCL) goes so far as to "hardwire" function calls such that if FOO calls BAR and FOO is compiled then if BAR is later redefined and FOO isn't, FOO will continue to call the old version of BAR, thereby destroying much of the flexibility of the Lisp.

We chose 4 settings of SPEED and SAFETY to study:

1. The default setting that the Lisp system has when it is initialized. This is what most people use.
2. SPEED 3, SAFETY 0 (written (3, 0) below) which should generate the fastest code.
3. SPEED 0, SAFETY 3 (written (0, 3) below) which should generate slow but very debuggable code, since the compiler should have done very few, if any, optimizations.
4. SPEED 3, SAFETY 2 (written (3, 2) below) which should generate optimized code while retaining "sanity checks".

The BB1 system used in these tests has very few declarations and does little numerical work. Both of these attributes seem common among most Common Lisp programs we use.

Code	Default	(3, 0)	(0, 3)	(3, 2)
Exp2	27	25	27	25
Exp2+	17	17	18	18
4/260	56	46	47	46
4/280	34	34	48	34
F-4/280	56	56	56	54
386	47	47	52	47
386T	54	54	60	54
mX	33	34	34	30
Maci	129	129	130	130
Sym	111	109	110	111
3/260	62	62	69	62
RT	75	76	77	75
DEC-III	63	60	71	70
Exp1	87	87	90	83
3/60	73	72	76	72
L-3/75	90	90	127	90
E-3/75	211	215	206	206
K-3/75	96	165	147	88
HP	115	113	141	118
DEC-II	207	206	231	236
XCL	559	543	559	556
Mac2	254	258	261	259

Table 5: BB1 Run Times for Various OPTIMIZE Settings

Table 5 and Figure 6 give the results for running BB1 with the four OPTIMIZE settings. Figure 7 shows the compilation times for the various OPTIMIZE settings.

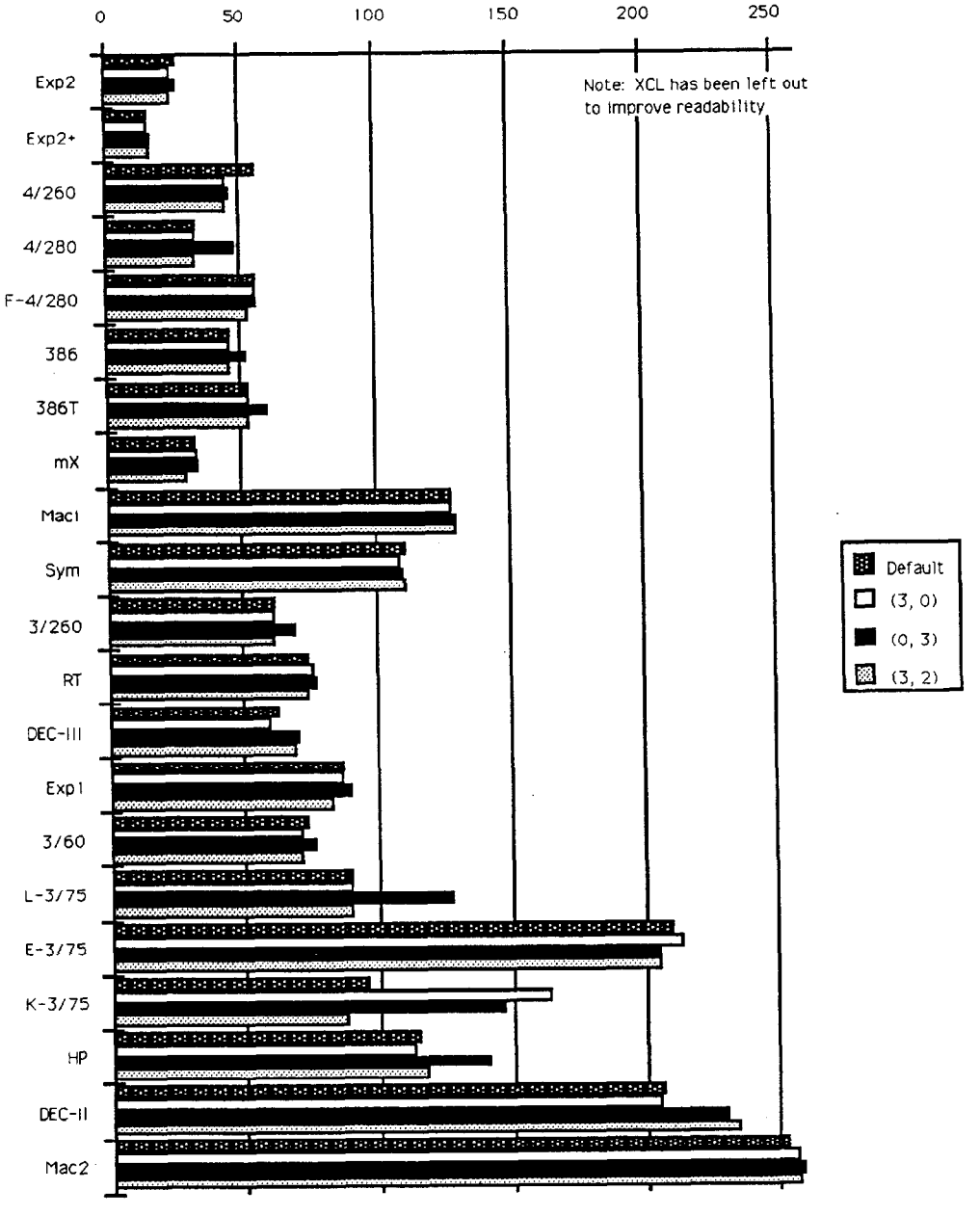

Figure 6: BB1 runs with various OPTIMIZE settings (sec)

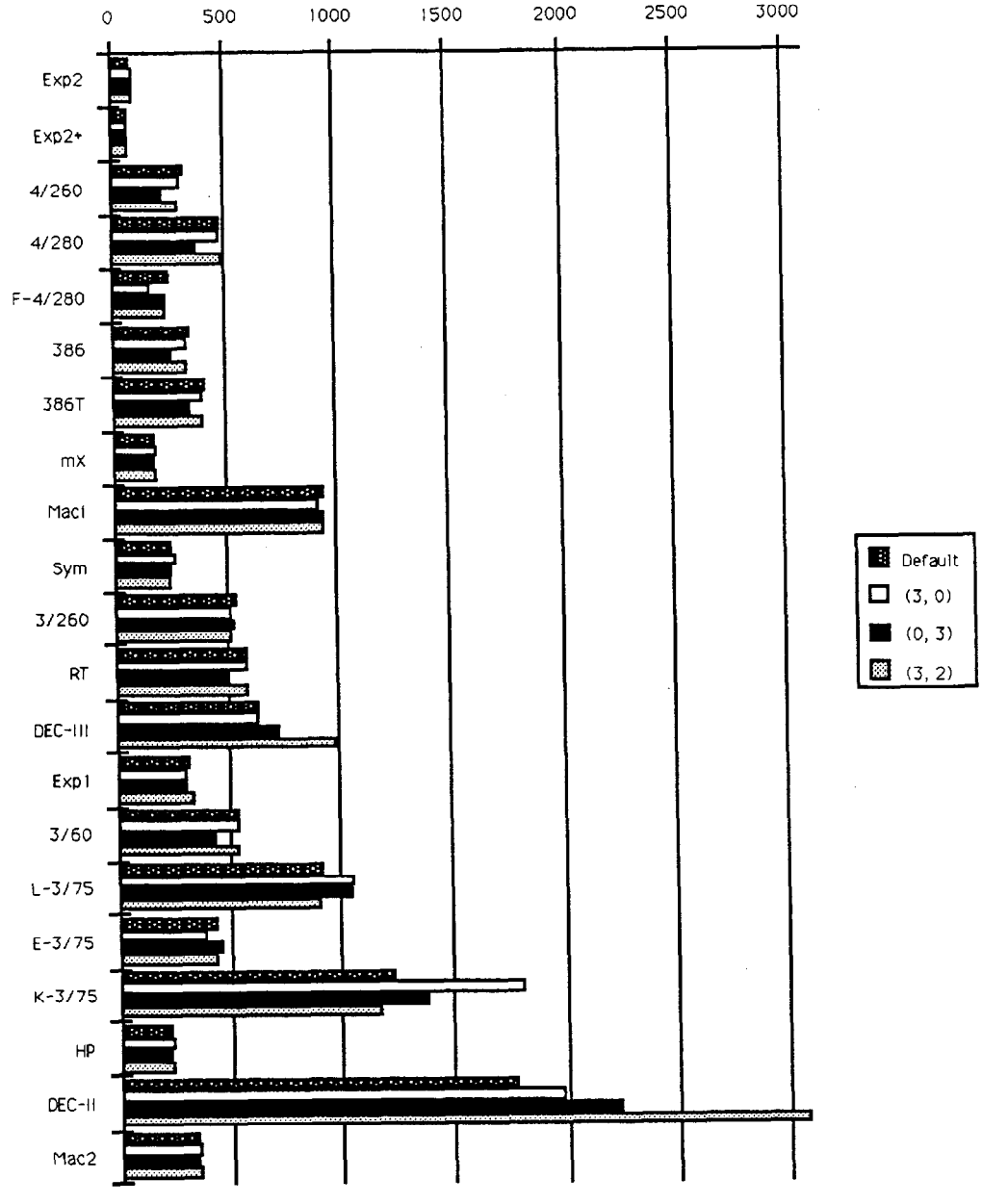

Figure 7: BB1 compilation times with various OPTIMIZE settings (sec)

These charts reveal somewhat surprising results. In several cases, SPEED 3, SAFETY 0 did not give the best results! Lucid Lisp did consistently better when SPEED was higher than SAFETY, as did the HP 9000, and VaxLisp. KCL was definitely behaving strangely with SPEED 0, SAFETY 3 coming out

a good bit faster than SPEED 3, SAFETY 0, with both of those much slower than "default" or SPEED 3, SAFETY 2.

Figure 8 depicts the speedup factor between the slowest time and the fastest time for the BB1 tests with various OPTIMIZE settings.

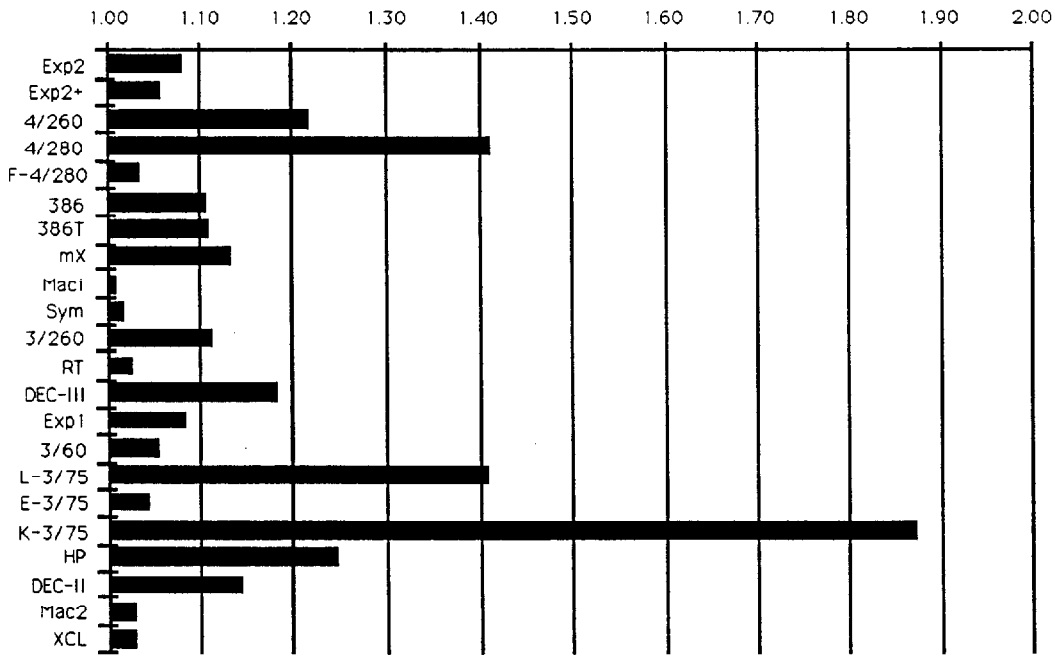

Figure 8: BB1 Speedup Factors Due to OPTIMZE Settings

8. Effect of Output Reduction on SOAR

The eight-puzzle benchmark for SOAR was originally written when SOAR ran primarily on slower machines than those tested here. Thus it tends to generate a lot of output relative to the amount of computation for some of the modes. For some systems, particularly those with large bit-mapped displays and full-screen windows, this output can be very expensive. To understand the extent of this effect we tested SOAR in the A mode and in the C mode both with full output, and with greatly reduced output (no trace). Table 6 with Figures 9 and 10 show results of these runs. Figure 11 depicts the amount of speedup (ratio of run times) realized by SOAR with reduced output.

	Mode A		Mode B	
Code	Full	Reduced	Full	Reduced
Exp2	33	18	18	16
Exp2+	23	11	13	11
4/260	23	13	11	9
4/280	35	15	14	11
F-4/280	36	36	20	19
386	41	27	21	19
386T	52	31	26	23
mX	50	27	29	27
Maci	165	65	63	44
Sym	55	40	34	32
3/260	49	33	23	22
RT	61	36	32	28
DEC-III	95	76	95	92
Exp1	90	63	75	71
3/60	66	38	34	31
L-3/75	82	67	45	41
E-3/75	124	109	81	80
K-3/75	186	136	120	111
HP	61	51	52	52
DEC-II	351	283	390	401
XCL	473	390	243	232

Table 6: SOAR Run Times with Full and Reduced Output

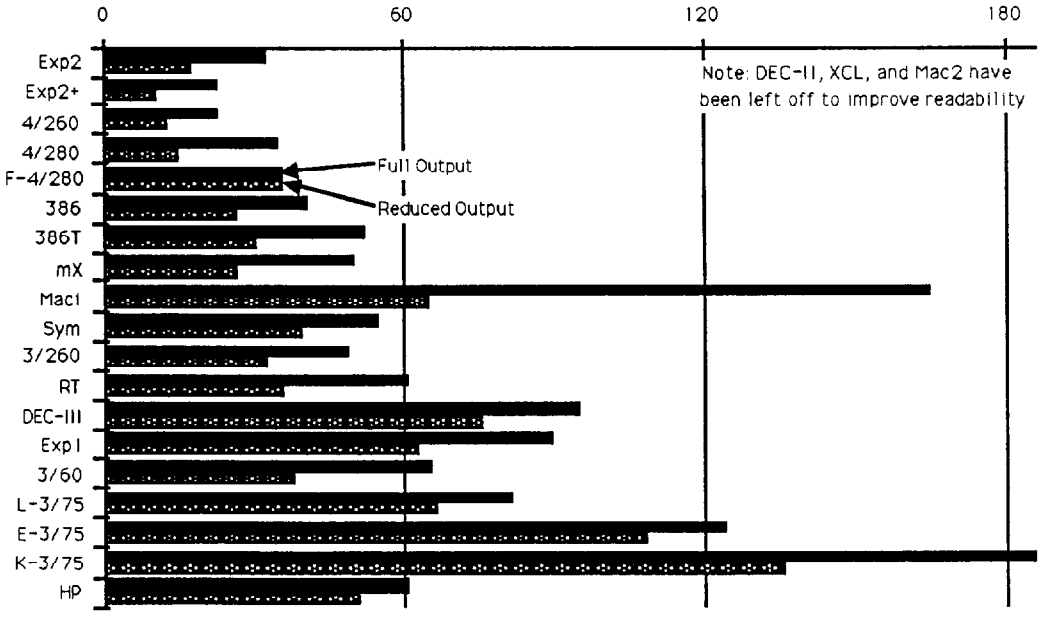

Figure 9: SOAR A Mode (sec)

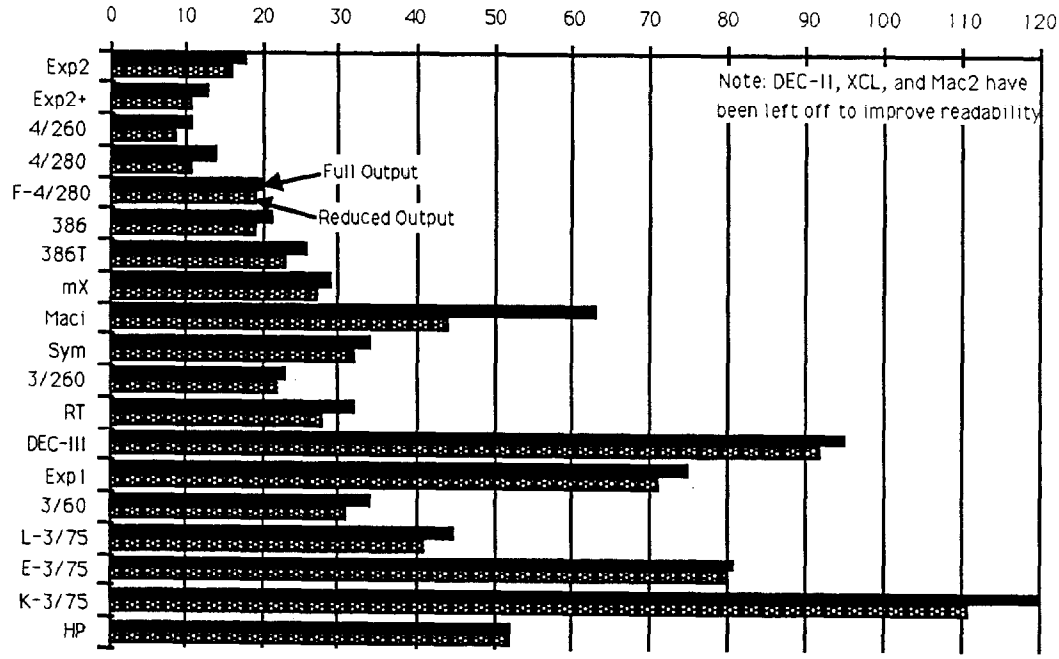

Figure 10: SOAR C Mode (sec)

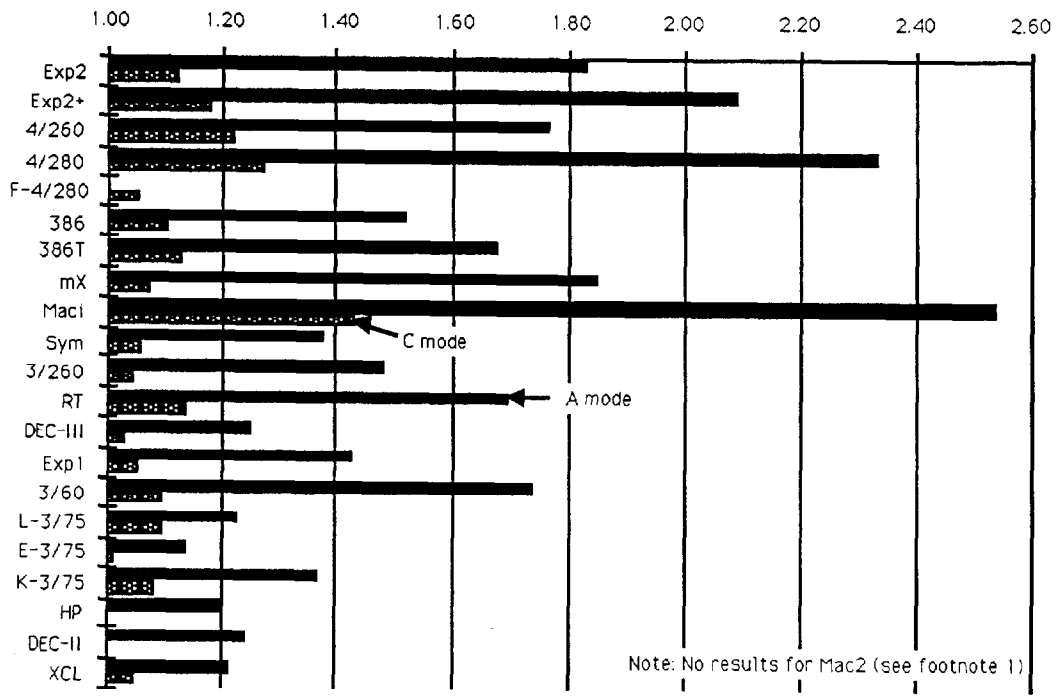

Figure 11: SOAR Speedup Due to Reduced Output

Three factors seemed to influence the speedup with reduced output:

- A fast processor, since the amount of time spent computing versus doing I/O would be reduced, causing a reduction in I/O time to be more significant.
- A larger screen or window since it is expensive to scroll a large area.
- A large-overhead I/O system such as the MacIvory's Dynamic Windows.

9. Future Work

Obvious areas in which this work might be extended include:

- Updating the results to reflect more recent versions of the Common Lisp systems;
- Adding more test systems, especially mainframes;
- Benchmarking other programs besides SOAR and BB1;
- Evaluating the effect of declarations on run times;
- Adding measurements of storage management overhead;
- Collecting more data on I/O overhead;
- Understanding better why platforms vary in performance from application to application and Lisp implementation to Lisp implementation.

10. Conclusions

Two moderate-sized applications, SOAR and BB1, were benchmarked on 22 Common Lisp systems to help in the evaluation of different Common Lisp systems. The run and compile times for these benchmarks were presented and discussed. A large variation was observed between the ranking of systems when running the SOAR test versus the ranking when running the BB1 test. This leads us to conclude that while these experimental results and ones like them can be used to class machines together roughly, it is impossible to use such a set of benchmarks to decide in advance how a given application will perform on a given system. There is no substitute for actually running the program on the systems in question.

Figure 12 shows the average of the normalized[1] run times for the test programs with the systems ranked in order. On the basis of this data, the systems tested may be ranked as follows:

---

1 The data were normalized by dividing each by the average of the results for all the tested implementations.

Very Fast (≤ 0.50 anr -- averaged normalized run time): TI Explorer II Plus (Exp2+), TI Explorer II (Exp2), and Sun 4 with Lucid Lisp (4/280 and 4/260)

Fast (> 0.50 anr, ≤ 1.00 anr): TI microExplorer (mX), Compaq 386 (386), Sun 4 with Franz Lisp (F-4/280), Compaq 386 portable (386T), Sun 3/260 (3/260), IBM RT/APC (RT), and Sun 3/60

Medium (> 1.00 anr, ≤ 1.50 anr): Symbolics 3645 (Sym), Sun 3/75 with Lucid Lisp (L-3/75), HP 9000/350 (HP), TI Explorer I (Exp1), and DEC MicroVax III (DEC-III)

Slow (> 1.50 anr, ≤ 2.50 anr): Symbolics MacIvory (Maci), Sun 3/75 with Kyoto Common Lisp (K-3/75), and Sun 3/75 with old Franz Extended Common Lisp (E-3/75)

Very Slow (> 2.50 anr): Apple Macintosh II with Allegro Lisp (Mac2), DEC MicroVax II (DEC-II), and Xerox 1186 (XCL),

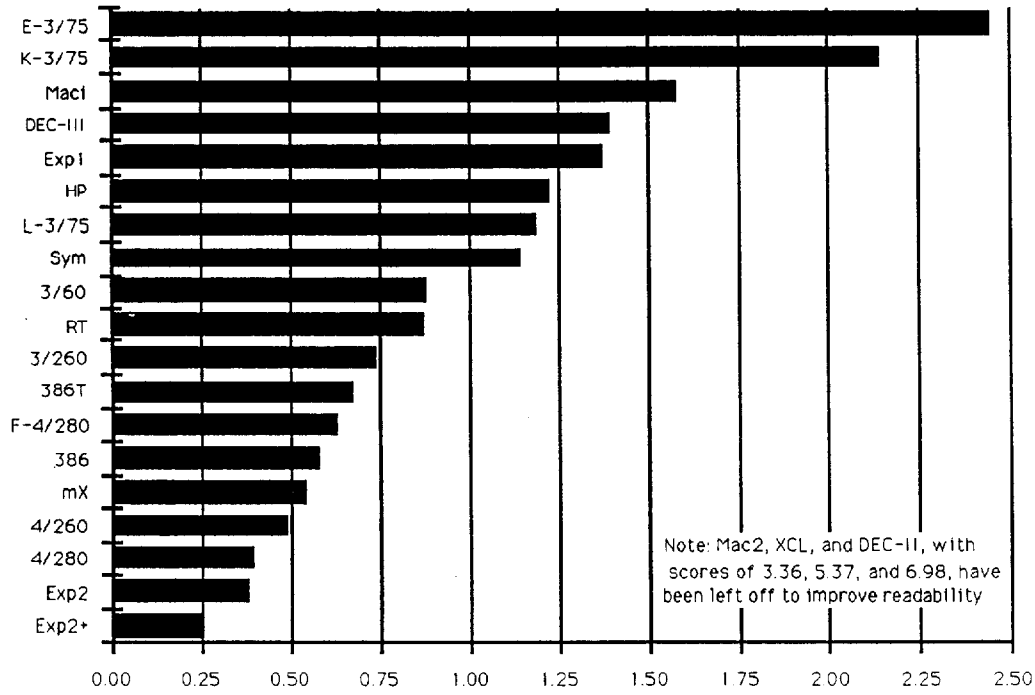

Figure 12: Averaged Normalized Run Times

We were surprised at the high speed of the small 386 machines, and at the slowness of the still early MacIvory, the DEC machines, and the Xerox machine.

Dedicated Lisp machines compile relatively faster than conventional machines, and, generally, conventional machine systems that took more time to compile produced faster code, as one would expect.

While the experiment to measure the effect of different settings of the OPTIMZE declaration was interesting, with such a small sample no real conclusion about the effect of various OPTIMIZE settings can be drawn. However the indications are that, in the absence of other declarations (e.g.. for TYPE), only relatively small gains are available. It is probably best to experiment with various settings to see which gets the best speed for a given program.

Reducing the amount of output that a program generates can have a large effect on the run time of the program, especially when moving the program to a faster machine. This indicates that it is worth taking some time to consider the nature of the I/O system and interaction needed by a program when designing a user interface for a fast-running program.

These results must be used very carefully since they represent only one piece of information about the performance of the very complex systems tested. We have measured only execution speed, but many aspects of the software will impact the development of programs such that in a given amount of time a program might be written for one machine that runs faster and perhaps with fewer errors than a program written in the same amount of time on another machine that ranks faster in these tests due to superior support given to the programmer during development. Do not underestimate the power of the programming environment.

11. Acknowledgements

This work would have been completely impossible without the assistance of many people and companies. Mike Kramer of Texas Instruments Inc. supplied the Explorer II Plus processor board. Eric Warner and Michael Borke of Sun Microsystems Inc. supplied access to the Sun 4 systems and the Sun 3/260 and 3/60 systems. Franz Inc. supplied a test version of Extended Common Lisp. Marty Hollander of Franz Inc. supplied a version of Allegro Common Lisp for the Sun 4. Jeff Harvey of Digital Equipment Corp. arranged access to the MicroVax systems. Susan Rosenbaum and Eric Gilbert of Lucid Inc. supplied access to the Compaq machines and the IBM RT. Bruce Hamilton of Hewlett Packard Inc. arranged access to the HP 9000. Many thanks to all of them.

12. References

[Gabriel 1985] Gabriel, R. P. **Performance and Evaluation of Lisp Programs**, M.I.T. Press, Cambridge, Massachusetts, 1985.

[Hayes-Roth 1985] Hayes-Roth, B. *A Blackboard Architecture for Control*, in Artificial Intelligence Journal, Volume 26, pp. 251-321, July 1985.

[Hayes-Roth 1988] Hayes-Roth, B., and Hewett, M. *BB1: An Implementation of the Blackboard Control Architecture*, in **Blackboard Systems**, edited by Robert Engelmore and Tony Morgan, Addison-Wesley, 1988, pp. 297-313.

[Laird 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. *Soar: An Architecture for General Intelligence*, in Artificial Intelligence Volume 33, Number 1, pp. 1-64, 1987.

[Steele 1984] Steele, G. L. Jr. **Common Lisp the Language**, Digital Press. 1984

Appendix A -- System Descriptions

This appendix contains detailed descriptions of the systems used in these measurements. In the descriptions, "Code" refers to a short name used to indicate the systems under test. Usually it is the model of the machine except where there is more than one Lisp for a machine (as in the case of the Sun 3/75) in which case a letter is prefixed to indicate the Lisp being used. "Timing Template" indicates how the information reported by the TIME macro was recorded. "Elapsed" indicates the total elapsed time, "run" indicates CPU time used, "gc" indicates time spent in garbage collection, "user" and "system" distinguish between user mode and kernel mode time, and "paging" indicates time waiting for virtual memory disk operations.Code:

Code: **3/260**
Computer Type: **Sun 3/260**
Operating System: **Sun OS 3.4**
Lisp: **Lucid 2.0**
Disk Configuration: **280MB**
Swapping Size: **60MB**
Memory Configuration: **8MB**
Display Configuration: **Color in mono mode**
Other Configuration:
Special Comments: **used :EXPAND 130 :GROWTH-RATE 130**
Timing Template: **elapsed (user-run + system-run)**
Date-of-test: **Summer 1988**

Code: **3/60**
Computer Type: **Sun 3/60**
Operating System: **Sun OS 3.4**
Lisp: **Lucid 2.1**
Disk Configuration: **SCSI 141MB**
Swapping Size: **unknown**
Memory Configuration: **24MB**
Display Configuration: **Hi Res Color in mono mode**
Other Configuration:
Special Comments:
Timing Template: **elapsed (user-run + system-run)**
Date-of-test: **Summer 1988**